Language neutral bindings for HLA

Philip A. Smith
Michael Fraser
David H. Stratton
University of Ballarat
University Drive
Mt Helen, 3353
Victoria
Australia
61-353-279237, 61-353-272729
p.smith@ballarat.edu.au, m.fraser@ballarat.edu.au, d.stratton@ballarat.edu.au

keywords: bindings, RTI, interoperability

Abstract: *The concept of an HLA binding is of a set of libraries and procedures which enable a program written in a given target language (such as Java or tcl) to communicate with an RTI (typically written in C++). Generation of HLA bindings is a non-trivial task which must be repeated for each language for which bindings are required. This paper describes bindings to the HLA which use sockets. This implementation decouples the target code from the code required to invoke functions on the RTI. This decoupling simplifies the generation of bindings for any language which can use TCP sockets. This paper describes these bindings with particular reference to an implementation of HLA bindings for the target language tcl.*

1. Introduction:

One of the goals of HLA is the incorporation of federates written in different languages and implemented on different platforms into interoperating federations. Achievement of this goal helps increase the potential of reuse of these federates into other federations.

Particular simulations may provide HLA capabilities. This is true of US military simulations for which HLA compliance is mandated. Other simulations, such as legacy code predating the HLA, are not capable of interacting with an RTI and therefore are not capable of interacting with other federates.

RTIs are often written in C++. Federates written in other languages require some way to be able to interact with the C++ API. Language bindings fulfil this purpose. HLA bindings have been written for a few languages including Java, C++ and ADA for certain RTIs. For other languages, including tcl, HLA bindings have not been developed to the authors' knowledge.

The development of language bindings for the HLA is not a trivial task. Furthermore the process must be repeated for each different language. This paper discusses the use of sockets in an attempt to simplify the process.

2. Background

Since 2003, the Distributed Simulation Laboratory (DSL), based at the University of Ballarat has been part of a consortium assembled for a project for the Australian Defence Force. This project, ADAPT, is designed to model whether humans will fit into aircraft crewstations and be able to perform given tasks. The details of this project are not the focus of this paper and will be expanded upon in other forums.

However, in order to perform the human modeling for this project, the DSL has been making use of a sophisticated human modeling package called Jack [1]. This package enables data obtained from scanning real humans to be used to create models within a virtual Jack world. The human model can then interact with a virtual environment and relevant data extracted.

Jack presents a tcl API that allows programmers to write tcl scripts to manipulate the Jack environment. Such a script can create humans and other objects, animate individual segments such as an arm or a leg, allow different entities to interact with each other and many other tasks. While Jack can interact with other systems, such as scanners,

through import and export mechanisms, HLA provides a more dynamic form of interaction.

Our initial motivation for developing the bindings was to increase the usability of Jack. Jack is designed specifically for modeling humans and does it very well. However, there are other purposes, for example the modeling of buildings, for which it was not specifically designed. While tcl is a powerful language and could be used to provide this functionality, the principle of reuse dictates that we use existing programs wherever possible. Language bindings which enabled HLA functions to be invoked as tcl commands were created to address this issue. These bindings were created using sockets which decouple the federate code from the RTI code.

This paper will describe these bindings and show how they simplify the creation of bindings in other languages. We expect that the provision of such bindings to greatly enhance the applicability of artifacts such as Jack. The next section describes these bindings.

3. Description

HLA is a well-defined set of rules and interface specifications [2]. All communication between federates takes place through a piece of software called the RTI. Thus, in order, for a simulation to interoperate with other HLA compliant federates it must be able to communicate with the RTI. While many RTIs have been developed, both commercial and open-source the RTI that we chose for our bindings was the RTI-NG developed by DMSO.

3.1 Architecture:

The architecture that is commonly associated with HLA is shown in Fig 1. In this architecture, the federate communicates with the RTI via two ambassadors – an RTI Ambassador and a Federate Ambassador. This architecture works well if the federate and the RTI are written in the same language. However, if the federate and the RTI are written in different languages then some sort of language bindings are required. The bindings described here use sockets which effectively split the ambassadors in two. The various components of the architecture can be seen in Fig 2. The components are:

Legacy code: The original simulation. Thus in our original example the legacy code is the non-HLA compliant Jack human model.

Target Federate: The domain specific code, written in the target language, that conducts an HLA session on behalf of the legacy code. In our example, the federate would be written in tcl and would be executed by the tcl interpreter built into Jack. It calls the RTI Ambassador functions (such as createFederationExecution) that are required for the HLA session. The Federate and the Legacy code are not parts of the actual bindings.

Bindings: The bindings themselves are made up of two sets of proxies. There are two proxies on the federate side which are written in tcl and two on the

General architecture showing a federate and RTI

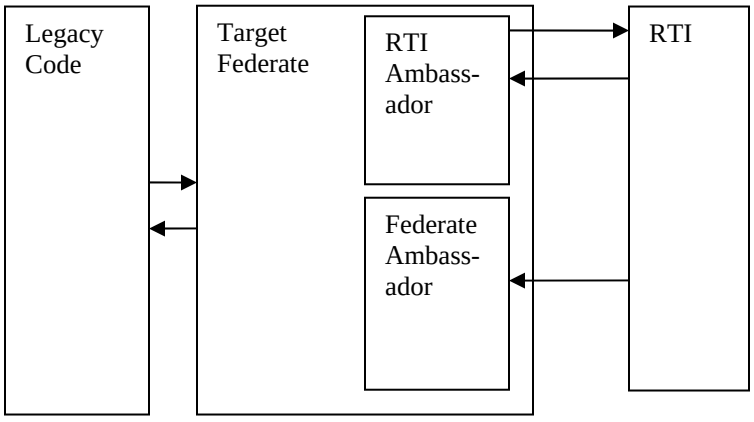

Fig 1

RTI-side which are written in C++. The two proxies on each side correspond to the RTI Ambassador and the Federate Ambassador and communicate with each other via sockets possibly across a network.

Federate side RTIAmbassador proxy: This is generic code that is not bound to any specific real world domain and consisting of tcl procedures corresponding to the RTI Ambassador functions. Each tcl procedure creates a string which describes the RTI function called from the target federate. The string consists of the function name followed by any arguments. The string is then passed via a socket the corresponding on the RTI-side. This proxy will also accept strings via the socket which represent return values from the function calls. The proxy function will interpret these values and return them to the federate.

RTI side RTIAmbassador proxy: The RTI side proxy, which is written in C++, accepts the string from the socket and parses it. The correct function call is then made directly on the RTI. Any return value (often a handle of some sort) is returned as a string to the federate side proxy via the socket. Like the federate side proxy, this code is non-domain specific.

RTI side Federate Ambassador Proxy: When the federate calls the tick function, the RTI has the opportunity of making federate ambassador callbacks to the federate. The RTI side federate ambassador proxy, encodes this function as a string and passes it back to the Federate side federate ambassador proxy as a string.

Federate side Federate Ambassador Proxy: This proxy captures the string returned from the RTI side and invokes the appropriate federate ambassador function. There is no need to parse this string because the tcl command eval can be used evaluate the string directly. Notice in the diagram there are no arrows representing return values from these functions. These were left out because most of the federate ambassador functions seem to be void. Unlike the other three proxies this code is domain specific and must be supplied by the creator of the federation.

The RTI. For our first set of bindings, the RTI used was the RTI NG. The RTI must be running and the RTI Server must be able to find it for the bindings to work correctly. The RTI is not part of the actual bindings.

The socket connection increases decoupling between the federate side and RTI side. This makes it possible to replace the federate side proxies without affecting the RTI side proxies. Thus it would possible to replace the tcl proxies on the federate side with (say) proxies written in Perl or other languages which provide TCP sockets. Similarly it would be possible to replace the RTI side components without affecting the federate side components. The tcl federate side proxies would be usable with an RTI side that used a different RTI for instance. This decoupling also enables the federate side components to be run at a location remote to the RTI side components. The RTI side proxies hide inconsistencies in RTI implementation code from the client. Thus it would not be necessary for the client code to know what RTI it was communicating with.

4. Usability issues

The aim of the bindings is to enable federate side proxies to be written in any language (that can use TCP sockets) so that they can interact with the same RTI side proxies. Thus all federate side proxies must generate strings to pass across the socket connection in a consistent manner. Some usability issues that might result are now described.

HLA defines many overloaded functions in its API. For example updateAttributeValues can have three arguments or four arguments depending upon whether or not it makes use of a timestamp. Furthermore, the two versions of updateAttributeValues have different return types – the version with the timestamp returns an eventRetractionHandle while the other version is void.

Function overloading is not supported in tcl. However, it does support default arguments. Thus the developer of a tcl federate could call updateAttributeValues with three or four arguments as appropriate. The Federate Side RTI Ambassador Proxy will generate a string with the correct number of arguments which is passed across the socket connection. This string is parsed on the server side and the appropriate version of updateAttributeValues is called. In tcl, the use of default arguments is appropriate because all overloaded functions differ in the number of arguments. It would be more difficult if the arguments differed only in type.

Bindings for a tcl federate and a C++ RTI using sockets

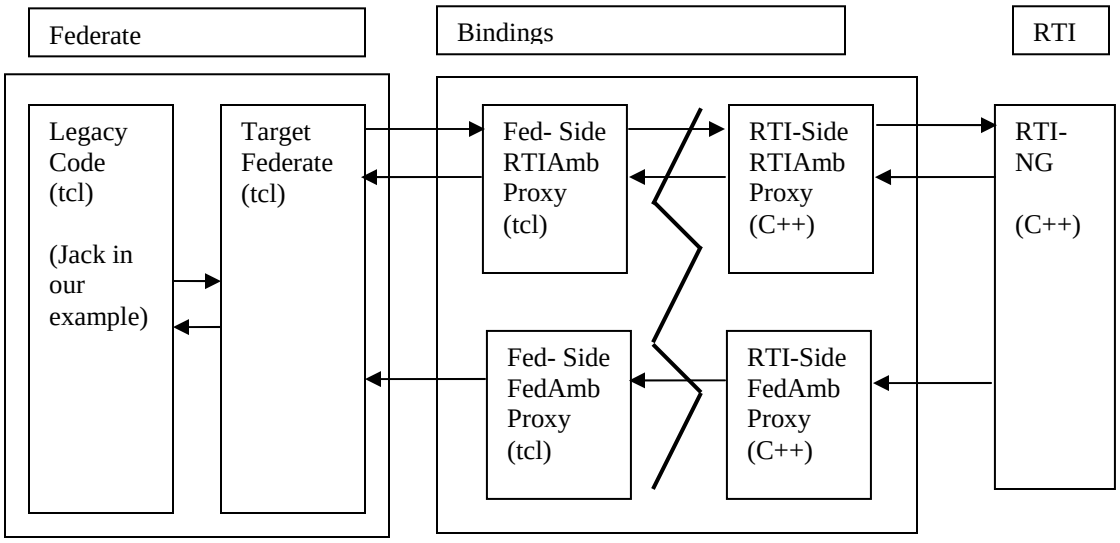

Fig 2

On the RTI side, the return value is encoded and returned via the socket to the federate side. If the function was invoked with four arguments, then the return value is the eventRetractionHandle. The Federate Side Federate Ambassador proxy must interpret this value and return it to the federate. If the function was invoked with three arguments then no value is returned.

A similar problem occurs in some federateAmbassador functions. For example, reflectAttributeValues, which is the callback for updateAttributeValues, can have three or five arguments depending on whether the original updateAttributeValues was called with a timestamp. The callback is encoded as a string with the correct number of arguments.

Some languages do not have function overloading or default arguments capabilities. In this case, there are two possibilities. One possibility is for the function names to be mangled in some way, perhaps updateAttributeValues and updateAttributeValuesWithTime. An alternative is for a placeholder, such as NULL, to be placed in the parameter list. The federate side RTI ambassador proxy should still generate the encoded string in a manner that entirely and precisely represents the defined HLA API standard. This enables the same server to be used for different federate target languages.

Another usability issue occurs when considering multiple executions. In this case, pairs of federate ambassador and RTI ambassador objects must be maintained. The use of sockets would cause problems in maintaining the associations between these objects when data is transferred. To overcome this problem, the name of the federation execution is passed with each RTI ambassador call as the first field of the string. This string is used on the RTI side as the first field in any federate ambassador callback. Note that, in the tcl implementation, only one Federate Side Federate Ambassador function exists for each callback and the federation name must be tested to enable different behaviours.

Another issue is that tcl is relatively typeless. Its main type is the string. All other types are pointers. This means that there is the problem of converting strings to other types used by the RTI. The main types that concerned us were:

- int – mainly used for HLA handles
- double – used for time values
- arrays of strings – it was necessary to convert tcl strings into char* arrays when using services such as updateAttributeValues.

It is necessary for the federate side to interpret these values correctly when they are passed along the wire.

Exceptions are generated by the RTI to signify unexpected or error conditions. The server encodes any exception as a string and sends it back to the client via the socket. The client recognizes the string as an exception and uses the tcl exception handling facilities to handle it appropriately.

Other exceptions are generated by federate ambassador functions. These have not been implemented as yet and have not been represented in the model in Fig 2.

Byte arrays: One technical issue which caused some concern was how to transmit byte arrays. Both the HLA 1.3 and IEEE1516 standards require attribute and parameter data to be transmitted as opaque byte arrays. This posed a problem as certain bytes in an opaque byte array can represent special characters such as end of line, carriage return and interrupt the socket transmission. While switching the socket to a binary stream mode could potentially overcome this problem, such socket functionality is not available on all platforms.

The chosen solution was to encode the data into an alphabet of printable characters. This method is commonly used in such situations as transmitting binary attachments through email, and posting to newsgroups etc. The two most recognised standards are the UUencode (UNIX to UNIX encoding) system and the MIME (Multipurpose Internet Mail Extensions) encoding system.

The base64 subset of MIME (RFC 3548) was chosen to encode binary transmissions as there is a large amount of freely available client libraries that conform to the standard, and the algorithm is fairly simple to implement if such libraries do not exist for the target platform. The base64 method works on 24 bit groups, splitting them into four individual 7 bit subgroups which are translated into a single character from an alphabet of 64 printable characters. Consequently four characters are used to represent every three bytes of the payload, making the encoded form 33% larger than the raw form.

For transmissions to the RTI, the raw data is encoded automatically by the client platform RTI ambassador. When received by the C++ socket server, the data is then decoded into raw binary form (an array of chars) and becomes a parameter to the appropriate RTI ambassador call. Likewise, for transmissions to the Federate the binary payload is converted into base64 form before it is transmitted to the target platform.

5. Direct invocation of RTI functions

While we used sockets to develop the bindings there are alternatives. Many languages, including tcl, have inbuilt means of invoking C++ functions directly. The use of these facilities can be time-consuming and error-prone. For tcl and several other languages, this process can be automated by a code-generation tool called Swig [3].

The use of Swig requires the declaration of the required RTI Ambassador functions in an interface file. This interface file is compiled by Visual C++ into a C++ file containing bindings. This C++ file is compiled, together with the actual implementation of these functions into a dll file. This dll file is then loaded into the tcl interpreter which can then call the C++ functions as if they were tcl commands.

Direct invocation of the RTI functions using Swig has the drawback of being tied to specific versions of tcl. While the use of Swig has benefits, it was the clear and unambiguous with of the creators of Jack that sockets be used. Their main concern was that of “version skew”. Jack, like many legacy systems, is large and hugely complex. The introduction of code which heavily relies on a particular version of a language can represent a great risk. The use of sockets and strings represents a far less risky proposition.

A further drawback is that Swig provides bindings for a limited number of languages, whereas the sockets can be used for a far greater number of languages.

A model of the direct invocation architecture is show in Fig 3.

6. Future Directions

The first task is to test the performance of the socket bindings. Very preliminary tests indicate that direct invocation shows better performance if the RTI and the federate are on the same machine. However, if the RTI is on a different machine and connected to the federate via a local network then the socket and direct invocation versions show similar performance.

- More exhaustive testing is necessary, however. If the results are favourable then one of our goals will be to create bindings in other languages such as Perl. Before deciding on new target languages an

Bindings using direct invocation

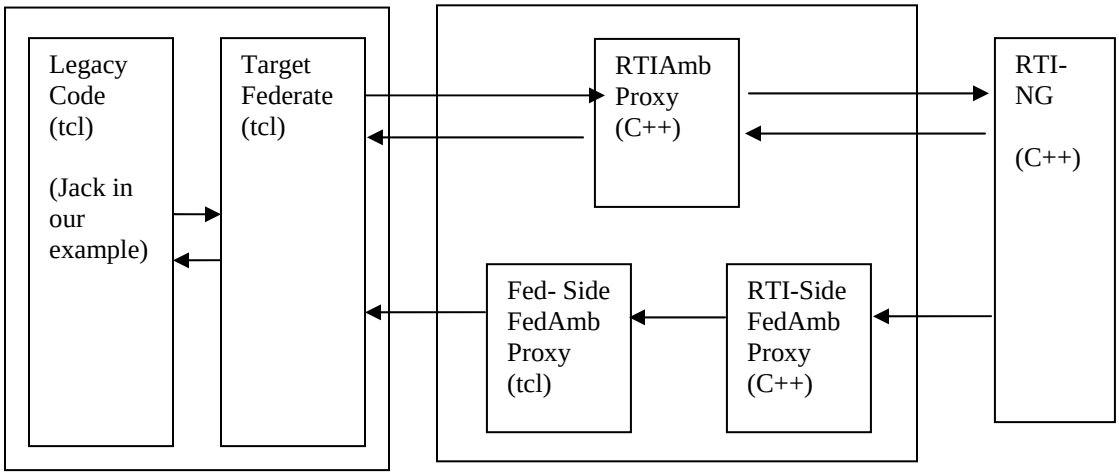

Fig 3

exhaustive search will be made to find applications that will benefit from HLA capability. A possibility might be CAD applications which require either LISP or VBA bindings.

Another of our goals is to use Jack as a federate in real-world applications. Possibilities of such applications might be:

- Modeling of ejection forces on a human body
- Modeling the effect of nuclear radiation on a human body
- Modeling of wind resistance on athletes.

None of these are easily modeled in Jack without HLA capability. The first task towards the completion of the above goal is the creation of a SOM for Jack. Intuitively, it would appear that a suitable SOM would follow the hierarchical organization of objects already present in Jack.

7. Conclusion

HLA was developed to facilitate the reuse of existing software artifacts and thus produce greater cost-effectiveness. However, there are software artifacts which might benefit from HLA but which are not capable of interacting with an RTI. This paper discusses a process of creating bindings, using sockets, which is designed to simplify the creation of such bindings. In particular, tcl bindings which we intend to use with a human modeling package called Jack were discussed. However, the ultimate aim is to enable other software artifacts, which are currently not used with HLA, to be drawn into the HLA net.

8. References

[1]http://www.ugs.com/products/tecnomatix/docs/fs_tecnomatix_jack.pdf . Jack human modeling and simulation: Virtual people, virtual places, real solutions, 2005.

[2] Kuhl, F., Weatherly, R.M., and Dahmann, J. “Creating Computer Simulation Systems: An introduction to the High Level Architecture”. Prentice Hall, 1999.

[3] http://www.swig.org/ Swig Home Page. 2006.

Author Biographies

PHILIP SMITH is a senior lecturer at the University of Ballarat and a founding member of the DSL. His interests are in programming and distributed simulation. He is currently involved in the ADAPT project with David Stratton.

MICHAEL FRASER is a research assistant at the University of Ballarat. He is currently employed as a programmer on the ADAPT project.

DAVID STRATTON is a senior lecturer at the University of Ballarat. He is a founding member and leader of the DSL and is responsible for organizing the software side of the ADAPT project. His interests are in networking and distributed simulation.